International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-13 Issue-12, November 2024

Generate High-Coverage Unit Test Data Using the LLM Tool

Ngoc Thi Bich Do, Chi Quynh Nguyen

Abstract: ***Unit testing is a critical phase in the software development lifecycle, essential for ensuring the quality and reliability of code. However, manually creating unit test scripts and preparing corresponding test data can be a time-consuming and labour-intensive process. To address these challenges, several automated approaches have been explored, including search-based, constraint-based, random-based, and symbolic execution-based techniques for generating unit tests. In recent years, the rapid advancement of large language models (LLMs) has opened new avenues for automating various tasks, including the automatic generation of unit test scripts and test data. Despite their potential, straightforwardly using LLMs to generate unit tests may lead to low test coverage. This means that a significant portion of the source code, including specific statements or branches, may remain untested, which can reduce the effectiveness of the tests. To overcome this limitation, the paper presents a novel approach that not only automates the generation of unit test scripts and test data but also improves test coverage. The proposed solution begins by using an LLM tool (such as ChatGPT) to generate initial unit test scripts and data from the source code. To enhance test coverage, the specification document of the source code is also input into the LLM to generate additional test data. Following this, a coverage checking tool is used to evaluate the test coverage and identify untested statements or branches. The LLM is then applied again to generate new test data specifically designed to address these gaps. The initial experimental results indicate that this method significantly improves test coverage, demonstrating its potential to enhance automated unit testing processes.***

Keywords**:** ***Branch Coverage, LLM, Python, Statement Coverage, Test Data Generation, Unit Test.***

I. INTRODUCTION

In the software development process, unit testing is an important activity to ensure code quality (i.e., that the code is error-free). One of the key criteria in generating unit tests is high coverage. Unit test scripts are considered to have high coverage if every statement (or branch) has at least one test case that runs through it. High coverage ensures that each statement (or branch) is tested, preventing software errors from being missed in statements (or branches) that lack test cases.

Generating unit tests is time-consuming and labour-intensive because it requires reading and understanding the source code, designing test data sets to ensure high coverage, and writing test scripts for each module that correspond to the test data. Therefore, much research has focused on automating unit testing [7].

One of the newly developed directions is the applicationn of large language models (LLMs). LLMs are advancing rapidly, and numerous studies are investigating their potential applications in automating tasks. In software development, LLMs have been utilised in various stages, including code generation and unit test generation. For unit test generation, LLM tools support generating unit test scripts from source code. Additionally, LLMs also assist in developing unit test scripts from specifications. Recent research results [1, 5, 11, 12] show the potential of applying LLMs in unit test script generation.

However, LLMs cannot independently evaluate the coverage percentage of the generated test scripts, nor determine how to add tests to increase coverage.

The paper focuses on automatically generating unit test data and scripts with high coverage by utilising two types of input: specifications and source code. By doing so, we can obtain two sets of test data instead of just one, providing more opportunities to increase coverage. Additionally, a coverage assessment tool will be applied to identify statements and branches that are not fully covered. Afterwards, the LLM will be used again to generate additional test data for these uncovered points.

II. RELATED WORKS

There are various well-established approaches for automating the generation of unit test problems, e.g., search-based, constraint-based, random-based, and symbolic execution- based methods [7].

Recently, several works have applied LLMs to test generation problems [1, 4, 5, 9, 11], with different purposes and/or in combination with other testing approaches. For example, [2] focuses on combining fuzz testing and LLMs; [3, 8] apply mutation testing with LLMs; [6, 12] focus on applying search-based testing with LLMs; [8] proposes a fine-tuned technique for the test suite generation problem; [9] proposed a method applying both a fine-tuned technique and a retrieval-augmented generation (RAG) technique for compiler validation.

[2] proposed a method to apply LLMs using both source code and specifications as inputs [2]. also applied LLMs multiple times using new fuzzing inputs and focused on mutation coverage objectives. Our work focuses on statement and branch objectives.

Manuscript received on 30 September 2024 | Revised Manuscript received on 19 October 2024 | Manuscript Accepted on 15 November 2024 | Manuscript published on 30 November 2024.

*Correspondence Author(s)
Ngoc thi Bich Do*, Faculty of Information Technology, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam. Email: ngocdtb@ptit.edu.vn, ORCID ID: 0009-0004-3250-0154
Chi Quynh Nguyen, Faculty, Department of Computer Science, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam. Email: chinq@ptit.edu.vn, ORCID ID: 0009-0007-6197-2486

[6] focuses on generating unit tests with high coverage objectives. To increase coverage results, [6] also creates prompts to generate more test cases for low-coverage functions. Our work identifies uncovered statements and branches rather than low-coverage functions to create prompts. Additionally, by utilising specification documents, our method has the opportunity to develop more test cases for parts of the source code that are not fully implemented.

[10] proposed a method to apply LLMs using both source code and documentation generated from comments. [10] generates unit tests for API functions with high coverage objectives. Our work also shares the high coverage objective. However, we also use both source code and documentation to create prompts for the LLM tool. Moreover, our work increases coverage results by identifying uncovered statements/branches in the source code and generating prompts focused on these statements/branches."

III. BACKGROUND

A. Unit Test Data and Unit Test Scripts

Unit tests are typically performed automatically by writing test scripts using unit test libraries specific to each programming language (e.g., JUnit for Java, CPPUNIT for C++). The programmer writes these scripts and runs them automatically. The test results then return either 'pass' or 'fail' for each test case (test data).

When conducting unit tests, the programmer needs to perform two tasks: (1) create test data sets that cover all statements (statement coverage) and branches (branch coverage) for each unit (method or class) of the original program; (2) write test scripts based on unit test libraries for the original program, using the test data created in step (1). Then, the test scripts will be run automatically, and the results will indicate if any cases have failed. Figure 1 and Figure 2 are examples of test scripts for a program that solves the quadratic equation $ax^2 + bx + c = 0$, corresponding to the test data set in Table 1. In Figure 1, each test function corresponds to a test case in Table 1. The assert statement checks whether the result of the solve_quadratic(a, b, c) function (the function solve quadratic equation $ax^2 + bx + c = 0$), using the input data a, b, and c from the input column (Table 1), matches the expected result in the 'expected Outputs' column (Table I). If the result is correct, running this test script will automatically return 'pass'; otherwise, it will return 'fail'. Another script style is represented in Figure 2, where each line is part of the self.test_cases represents a test case corresponding to a test data set (a row in Table 1). Then, only one test script will be applied for the test data list self.test_cases.

Table I: Test Data of Program Solves the Quadratic Equation $ax^2 + bx + c = 0$

Test Purpose	Test Data		
	Description	Input (a,b,c)	Expected Outputs
Two Real Roots	Discriminant > 0	(1, -3, 2)	(2.0, 1.0)
One Real Root	Discriminant = 0	(1, 2, 1)	(-1.0,)
No Real Roots	Discriminant < 0	(1, 0, 1)	None

```
import unittest
class TestQuadraticSolver(unittest.TestCase):
    def test_two_real_roots(self):
        result = solve_quadratic(1, -3, 2)
        self.assertEqual(result, (2.0, 1.0))

    def test_one_real_root(self):
        result = solve_quadratic(1, 2, 1)
        self.assertEqual(result, (-1.0,))

    def test_no_real_roots(self):
        result = solve_quadratic(1, 0, 1)
        self.assertIsNone(result)
```

[Fig. 1: Script Test of Function Solves the Quadratic Equation $ax^2 + bx + c$: Solve_Quadratic (a,b,c)]

```
import unittest
class TestQuadraticSolver(unittest.TestCase):
    def setUp(self):
        self.test_cases = [
            # (a, b, c, expected_roots)
            (1, -3, 2, (2.0, 1.0)),
            (1, 2, 1, (-1.0, -1.0)),
            (1, 0, 1, None),
        ]

    def test_solve_quadratic(self):
        for a, b, c, expected in self.test_cases:
            result = solve_quadratic(a, b, c)
            self.assertEqual(result, expected)
```

[Fig. 2: Script Test of Function Solves the Quadratic Equation $ax^2 + bx + c$: Solve_Quadratic (a,b,c)]

Two situations may prevent 100% coverage:

- **Situation 1:** The test data set is not comprehensive and may miss certain cases.
- **Situation 2:** The source code contains dead code (i.e., dead statements or dead logic). These are parts of the code that are never executed with any dataset.

In **Situation 1**, we can improve coverage by generating a better set of test data. In **Situation 2**, the coverage cannot be improved. Instead, the source code should be reviewed to identify and correct errors, as well as determine the cause of any existing dead code.

B. LLM and Application in Unit Test

Large Language Models (LLMs) are a type of machine learning model designed to process and generate natural language. Many people are adopting chat tools that utilise these LLMs and are gradually integrating them into daily life, work, and study to help reduce the time spent searching for information and solutions, as well as to automate certain stages in the creation of digital products (e.g., images, text, programming code).

In programming, LLM tools can be applied in various contexts, such as answering questions, generating source code, creating test cases, and unit test scripts.

When generating unit test scripts, LLM tools can assist in creating test scripts for the provided source code. These test scripts are written using the corresponding unit test library for each language and include descriptions for each script.

LLMs have the advantage of being able to generate test data for any program or specification. However, they still face some common issues:

- Issue 1: They cannot independently assess the quality of the test dataset, including evaluating coverage. As a result, there is a lack of direction in generating test data, which hinders the increase in coverage percentage.
- Issue 2: Some test data may have the same objective and could be redundant, requiring removal.
- Issue 3: Some test data may have incorrect results, including errors in the input, output, or scripts.

Issues 1 and 2 can be addressed by integrating a coverage evaluation tool to identify areas that are not covered. Issue 3 can be mitigated by running the test cases and reviewing the results, or manually reviewing the test set. Another approach to minimize Issue 3 is to provide a pre-defined template for inputs, outputs, and test scripts.

IV. PROPOSED METHOD OF APPLYING LLM FOR UNIT TEST GENERATION

A. Proposed Method

We have proposed a method for generating test scripts and test data to achieve high coverage objectives in unit testing. The required inputs for our method are specification documents and the corresponding unit source code. The LLM tool will be used several times to generate test scripts and test data with high coverage. Additionally, a coverage evaluation tool will be used to assess coverage results and identify any uncovered areas.

Algorithm 1 shows our proposed method.

Algorithm 1:
Input:
- sourceCode: source code of a test function/class
- spec: specification document for the above test function/class

Output: testScript1, testData, unCover points, %cover

```
1.  p1= createPromt1(sourceCode)
2.  (testData1, testScript)=ApplyLLM(p1)
3.  p2=createPromt2(spec, testData1)
4.  (testData2) = ApplyLLM(p2)
5.  testData= merge(testData1, testData2)
6.  (unCover, %cover) = findUncover(testData ,
    testScript, sourceCode)
7.  if %cover ==100% then
8.    return (testScript, testData, unCover,
    %cover)
9.  p3 = createPromt3(unCover, sourceCode,
    testData1)
10. (testData3) = ApplyLLM(p3)
11. testData = merge (testData, testData3)
12. (unCover, %cover) = findUncover(testData ,
    testScript, sourCode)
13. return (testScript, testData, unCover,
    %cover)
```

Line 1: Create the first prompt using the source code of the test function/class.
Line 2: Use the LLM to generate the test script and corresponding test data list for the source code.
Line 3: Create the second prompt using the specification document of source code and request that the output follows the format of the test data list generated in Line 2.
Line 4: Use the LLM to generate test data from the specifications in the correct format.
Line 5: Combine the test data sets generated from Lines 2 and 6. Evaluate coverage and identify uncovered statements or branches.
Lines 9 and 10: Create the third prompt for the uncovered areas and apply the LLM to generate additional test data.
Lines 11 and 12: Merge the test data sets and re-evaluate coverage.

B. Proposed Unit Test Script and Test Data List Template

There are several ways to generate test scripts: (1) each test case will generate a test function that contains a test scrip with corresponding test data (e.g. Figure 1); (2) all test data will be written in a list and a test script will be applied (e.g., Fig. 1).
Our method will use method (2) due to the following reasons:

- Unified Test Script: In Algorithm 1, the LLM tool is applied multiple times to increase coverage results. To ensure the generated test script is unified, the test script will be generated only the first time the LLM tool is called. This test script can be applied to the test data list.
- Unified Test Data List Format: After applying the LLM tool several times, the corresponding test data can be easily added to the test data list using the same format. Additionally, this test data list will be more easily managed (e.g., adding, deleting, modifying) compared to the approach in (1).

Figure 3 shows the proposed test script and test data template.

```
  import unittest
class <TEST CLASS NAME>(unittest.TestCase):
    def setUp(self):
        self.test_cases = [
            (<input data>, <expected output data>),
            …
        ]
    def <test function name>(self):
        for <input>, expected in self.test_cases:
            result = <function name>(<input format>)
            self.assertEqual(result, expected)
```

[Fig. 3: Proposed Test Script and Test Data List Template]

C. Proposed Prompts

In Algorithm 1, three kinds of prompts will be used:

- Prompt 1: The input is the source code of a function/class; the output is a test script and the first test data list; the prompt must reflect the purpose: to generate a test script and test data list using the template in Figure 3 with a high statement coverage objective.
- Prompt 2: The input is the specification document of the corresponding function/class; the output is the second test data list; the prompt must reflect the purpose: to generate a test data list that has the same format as the first test data list above, with a high specification coverage objective.
- Prompt 3: the input is uncovered statements/branches found in Algorithm 1 (line 7); the output is the third test data list; the prompt must reflect the purpose: to generate the test data list that has the same format as the first test data list above to cover uncovered statements/branches.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

The proposed templates for the above prompt types are shown below:

Prompt 1:

> You are an expert Python test-driven developer.
> For the source code below:
> *<source code of a test function/class>*
> Create a new test function, ensuring that it is correct and effectively improves coverage.
> Respond ONLY with the Python code using the following template.
> Remember that test data must be written in a list that includes: inputs and expected outputs.
> *<script and test data template from Figure 3>*

Prompt 2:

> You are an expert Python test-driven developer.
> The specification below:
> *<specification document corresponding to the test function/class>*
> Create test data that ensures coverage of both true and false cases in every condition of the above specification. The test data must use a table format that includes: inputs and expected outputs. Respond ONLY with test data using the following template:
> *<test data list template from Figure 3>*
> Examples:
> *<some lines of test data generated from Algorithm 1 (line 2)>*

Prompt 3

> For the source code
> <source code of the test function/class>
> still lacks coverage of the following conditions:
> *<uncover branches found in Algorithm 1 (line 6) with corresponding actual/false cases>*
> Add more test data for only the above conditions. Writing the test data list in the following template
> *<some lines of test data generated from Algorithm 1 (line 2)>*

V. EVALUATION

The experiment is executed using:

- LLM tool: chatGPT version GPT-4o (https://chatgpt.com/).
- Source code: the Python programming language.
- Unit test framework: the unittest (http://docs.python.org/library/unittest.html).
- Coverage tool: Coverage.py (https://coverage.readthedocs.io/en/latest/)

A. Case Study:

We demonstrate and evaluate our proposed method with the following case study:

B. Specification Document:

A positive integer is called a number that increases or decreases if the following conditions are satisfied:

- Has three or more digits
- Find a position in the sequence of numbers so that from the left to that position, the sequence satisfies the ascending (tightening) order and from that position to the end, it satisfies the descending (tightening) order.

Write a program to check if a number is increasing or decreasing. The number must be an integer with no more than 18 digits.
Input
A positive integer N with no more than 18 digits
Output
YES or NO, depending on the test results.
Example:
Input: 12342
Output: YES
Input: 23342
Output: NO
Input: 5678961
Output: YES

C. Corresponding Source Code:

```
def is_increase_decrease_number(N: str) -> str:
    # The number must have at least three digits
    if len(N) < 3:
        return "NO"
     n = len(N)
        # Find the "peak" point
    peak = -1
    for i in range(1, n):
        if N[i] < N[i-1]:  # Find first decreasing
point
            peak = i - 1
            break
        # If no peak point is found, the number doesn't
have both increasing and decreasing sequences
    if peak == -1 or peak == 0 or peak == n-1:
        return "NO"
        # Check if the left part is strictly
increasing
    for i in range(1, peak+1):
        if N[i] <= N[i-1]:
            return "NO"
        # Check if the right part is strictly
decreasing
    for i in range(peak+1, n):
        if N[i] >= N[i-1]:
            return "NO"
        return "YES"
```

D. The Test Script and Test Data List are Generated by Executing Prompt 1:

```
import unittest
class Test Is Increase Decrease Number(unittest. Test
Case):
    def setUp(self):
        self.test_cases = [
            ("12342", "YES"),
            ("23342", "NO"),
            ("5678961", "YES"),
            ("54321", "NO"),
            ("987654", "NO"),
            ("1456321", "YES"),
            ("111", "NO"), # Edge case: All digits are
the same
            ("12345678987654321", "YES"), # Long
number with valid increase-decrease pattern
            ("123454321", "YES"), # Symmetrical
number
            ("123456789", "NO"), # Strictly
increasing
            ("987654321", "NO"), # Strictly
decreasing
            ("10234567898765432", "YES"), # Zero
included, valid increase-decrease
            ("1001", "NO"), # Edge case: Minimum peak
at the start
```


Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

```
            ("121", "YES"), # Simple
increase-decrease case
            ("1221", "NO"), # Not strictly increasing
in the first half
        ]
    def test_is_increase_decrease_number(self):
        for num_input, expected in self.test_cases:
            result =
is_increase_decrease_number(num_input)
            self.assert Equal(result, expected)
```

E. The Test Data List is Generated by Executing Prompt 2:

```
self.test_cases = [
    # Valid increase-decrease cases
    ("12342", "YES"),
    ("5678961", "YES"),
    ("13579", "YES"),
    ("12345678987654321", "YES"),

    # Invalid cases (not three or more digits)
    ("1", "NO"),
    ("12", "NO"),

    # Invalid cases (not strictly increasing, then
decreasing)
    ("23342", "NO"),
    ("1233345", "NO"),
    ("987654321", "NO"),
    ("112233445566", "NO"),

    # Edge cases (exactly 18 digits)
    ("123456789012345678", "NO"),
    ("12345678987654321", "YES"
    ("123456789012345679", "YES"),
    ("999999999999999999", "NO"),

    # Mixed cases
    ("345678321", "YES"),
    ("1122334455", "NO"),
    ("111223344556677", "NO"),
]
```

F. The Test Data List is Generated by Executing Prompt 3:

After executing Prompt 2, one branch is still not covered if N[i]> `N[i-1]`. Thus, Prompt three will be executed.

```
self.test_cases = [
    # Cases where the right part is not strictly
decreasing (N[i] >= N[i-1] is true)
    ("1234554321", "NO"),       # Right part is flat
(repeated '5')
    ("1234543321", "NO"),         # Right part has
repeated '3', not strictly decreasing
    ("123450000", "NO"),          # Flat sequence
(repeated '0's) after the peak
…
]
```

G. Observations

We have also addressed several problems and obtained the coverage results, as shown in Table II.

Table II: Coverage Results

No.	Code Coverage Results Of the Test Data Generated by			Noted
	Prompt 1	Prompt 2	Prompt 3	
Case study	88%	94%	100%	Prompt 2 covers more specific cases.
Problem 2	100%	100%	No run	Prompt 2 covers more specific cases
Problem 3	90%	100%	No run	

Analyzing the coverage results in Table II and the generated test data, we have the following observations:

Pros:

- Executing the LLM tool with Prompt 2 (i.e., to generate test data from the specification document) can create test data that covers more statements and branches. For example, in the case study, test data ("1", "NO") and ("12", "NO") will cover the branch *if len(N) < 3,* with proper case.
- Executing the LLM tool with Prompt 2 (i.e., to generate test data from the specification document) can create test data that the source code does not process. For example, in the case study, test data `("123456789012345678", "NO")` is within specifications (1 ≤ N ≤ 10^18).
- Executing the LLM tool with Prompt 3 can generate test data to cover more statements and branches. For example, in the case study, test data `("1234543321", "NO")` will cover the branch `if N[i] >= N[i-1]:` with proper case.
- The test scripts are generated using the correct template.
- For every run of the LLM tool, the test data lists are generated using the correct template.

Cons:

- Some generated expected results are incorrect. For example, in a case study, test data are used. `("10234567898765432", "YES")` generated from Prompt 1, test data `("123456789012345679", "YES")` generated by Prompt 2
- Cannot generate test data for some uncovered statements/branches.
- Some generated test data is redundant; they do not contribute to improving coverage results.

VI. CONCLUSION

The proposed method for generating unit test scripts and test data aims to increase coverage using LLM. The method involves:

- Using both source code and specification documents for test function/class: The LLM generates test scripts by considering both the source code and the specification documents, ensuring more comprehensive coverage.
- Improving coverage results by generating additional test data: The test data can be expanded to increase coverage for previously uncovered statements or branches.
- Creating template prompts, test scripts, and test data: These templates guide the LLM in generating outputs, ensuring that the test script and test data list are consistent, efficient, and correctly formatted.

Future Directions:

- Reducing the test data list size by removing test data that does not contribute to coverage results.
- The current approach focuses on predefined unit tests but has not yet been applied to the entire source code.
- Solving the issue of incorrect expected results.

DECLARATION STATEMENT

After aggregating input from all authors, I must verify the accuracy of the following information as the article's author.

- **Conflicts of Interest/Competing Interests:** Based on my understanding, this article does not have any conflicts of interest.
- **Funding Support:** This article has not been funded by any

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

organizations or agencies. This independence ensures that the research is conducted with objectivity and without any external influence.

- **Ethical Approval and Consent to Participate:** The content of this article does not necessitate ethical approval or consent to participate with supporting documentation.
- **Data Access Statement and Material Availability:** The adequate resources of this article are publicly accessible.
- **Author's Contributions:** The authorship of this article is contributed equally to all participating individuals.

REFERENCES

1. Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and . Yin. “ChatUniTest: A Framework for LLM-Based Test Generation”. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (2024), pp 572–576. doi: https://doi.org/10.1145/3663529.3663801
2. X. Chun, M. Paltenghi, J. L. Tian, M.L. Pradel and L. Zhang. “Fuzz4ALL: Universal Fuzzing with Large Language Models”. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 126, 1–13. doi: https://doi.org/10.1145/3597503.3639121
3. A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais. “Effective test generation using pre-trained Large Language Models and mutation testing”. Inf. Softw. Technol. (2024), 171. doi: https://doi.org/10.1016/j.infsof.2024.107468
4. A. Deljouyi. “Understandable Test Generation Through Capture/Replay and LLMs”. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24). Association for Computing Machinery, New York, NY, USA, 2024, pp. 261–263. doi: https://doi.org/10.1145/3639478.3639789
5. W. Junjie, Y. Huang, C. Chen, Z. Liu, S. Wang and Q. Wang. “Software Testing With Large Language Models: Survey, Landscape, and Vision”, IEEE Transactions on Software Engineering 50, (2023), pp 911-936. doi: https://doi.org/10.1109/TSE.2024.3368208
6. C. Lemieux, J. P. Inala, S. K. Lahiri and S. Sen, "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models". 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, (2023), pp. 919-931. doi: https://doi.org/10.1109/ICSE48619.2023.00085
7. E. Daka and G. Fraser. “A Survey on Unit Testing Practices and Problems”. In Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE '14). IEEE Computer Society, USA (2014). pp. 201–211. doi: https://doi.org/10.1109/ISSRE.2014.11
8. J, Liu, C. S. Xia, Y. Wang, and L. Zhang. “Is your code generated by ChatGPT correct? rigorous evaluation of large language models for code generation”. In Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS '23). Curran Associates Inc., Red Hook, NY, USA, Article 943, (2023), pp 21558–21572. doi:https://doi.org/10.1016/j.future.2024.05.034
9. C. Munley, A. Jarmusch, S. Chandrasekaran, “LLM4VV: Developing LLM-driven testsuite for compiler validation”, Future Generation Computer Systems, Volume 160, (2024), pp 1-13, ISSN 0167-739X. doi: https://doi.org/10.1109/TSE.2023.3334955
10. M. Schäfer, S. Nadi, A. Eghbali and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation", in IEEE Transactions on Software Engineering, vol. 50, no. 1, (2024), pp. 85-105. doi: https://doi.org/10.1109/TSE.2023.3334955
11. Y. Shengcheng, C. Fang, Y. Ling, C. Wu and Z. Chen. “LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities”. 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) (2023), pp 206-217. doi: https://doi.org/10.1109/QRS60937.2023.00029
12. Y. Tang, Z. Liu, Z. Zhou and X. Luo, "ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation", in IEEE Transactions on Software Engineering, vol. 50, no. 06, (2024), pp. 1340-1359. doi: https://doi.org/10.1109/TSE.2024.3382365

AUTHORS PROFILE

In 2004, **Thi Bich Ngoc** successfully earned her Bachelor of Science degree in Information Technology from the University of Science and Technology of Vietnam. Subsequently, she successfully earned her Master's degree in Computer Science from the University of Hanoi in 2007. In 2010, she successfully earned a Ph.D. degree from the Japan Advanced Institute of Science and Technology, specialising in the field of Information Science. Since 2013, she has been an esteemed lecturer within the Faculty of Information Technology at the Posts and Telecommunications Institute of Technology. Her research interests are software testing, formal methods, numerical analysis, data mining, and machine learning.

Chi Quynh Nguyen graduated with a Bachelor of Science in Computer Science from Hanoi University of Technology in Vietnam in 1999, earning a summa cum laude distinction. She then received a Vietnamese Government Fellowship to pursue a Master of Science in Computer Science at the University of California, Davis, USA, in 2004. Then she became a Ph.D. candidate in Computer Science at the same University in 2006. Since 2008, she has been a senior lecturer in the Faculty of Information Technology at the Posts and Telecommunications Institute of Technology in Hanoi, Vietnam. Her primary research focuses on data warehousing, data mining, and bioinformatics, as well as Mobility prediction, self-configuration of MANets, and data aggregation methods in sensor networks.